UNIVERSITY OF GLASGOW

COMPUTING DEPARTMENT

MULTUM PASCAL COMPILER MEMO NO. 2 (21/1/73)

AMENDMENTS TO THE RUN TIME ENVIRONMENT

PROGRAM STRUCTURE

The following sets out a model for the structure of the SUL output by the Pascal compiler. Compared with our previous ideas, this model makes better use of the memory-held registers (MHRs), speeds up and shortens most procedure entry sequences, shortens code referring to large (>255) constants, and permits us to capitalise on the INTEGRATC (when it becomes available) to organise segmentation and overlay in the program. In the new model, the output consists of several <u>modules</u> of SUL, each module corresponding to

(1) a procedure, or
(2) the program part, or
(3) the global scalar space, or
(4) a table of the large constants.

Take the skeletal Pascal program below as an example.

```
const ......
type .......
var ........
:
:
:
value ......
procedure  P1
:
:
:
procedure  P2
:
:
:
begin {program part}..... end .
```

The SUL output code consists of:

the output must be in this order until the INTEGRATOR is made available

(1) a module for P1,
(2) a module for each of the other procedures or functions,
(3) a module for P2,
(4) a module containing a consolidated list of all large constants in the program,
(5) a module starting with the global scalar space and containing embedded constants corresponding to the <u>value</u> section - this module is the base of the stack.
(6) a module for the program part.

At the end of the global declarations the compiler can calculate the size of the global scalar space and hence the number of MHRs needed to address it, g say. MHRs (g+1) and on are dedicated to addressing the 'constants module'. The total number of these MHRs is not known until the end of the program is reached, but since no procedure ever needs to access a constant beyond those currently detected while it is being compiled, no problem arises. Suppose MHRs (g+1) through d, d<6, are needed altogether. Then the global scalar space module contains/

contains embedded address constants to set up M1 : Mg to span itself and M(g+1) : Md to span the constants.

THE GLOBAL MODULE

The global module will be integrated or assembled into a read/write segment containing the stack, of which it forms the base. In structure the global module is identical with any other stack frame. When files are implemented it will correspond to that of a procedure declared by:

procedure main (input, output, code file : text)
(where type text = packed file of char) but in the meantime the program, "main", will not have any parameters.

However, since the global stack frame has a special place in the program, it has properties not shared by other stack frames:

(1) The records and arrays which have been initialised in the value section (and probably all the others as well) will have to be pre-allocated.
(2) M1 through Mg, g<5, must be preset to span the global scalar space.
(3) M(g+1) through Md, d<6, must be preset to span the constants module.

The global module will be called "GLOBAL" in SUL, the program part will be called "MAIN", and the constants module will be called 'CNSTNT'.

PROCEDURE AND FUNCTION MODULES

The SUL module generated from a Pascal procedure (or function) declaration consists of three parts: a body, an exit sequence and an entry sequence. The new idea of this section is that they should be output in that order, with the entry sequence last. This allows all the data gathered about the procedure (or function) during its translation to be used in optimising the entry sequence.

(1) There is no need to set up the static pointer if the procedure references only locals and parameters.
(2) There is no need to set up MHR i if the procedure does not reference the global or constants area spanned by Mi.

(Note that the definition of the term "references" is recursive - a procedure references an object if it uses it in its statements or it contains the declaration of procedure which references it. As a result, these optimisations can hardly ever be performed at the outer lexical levels, but can usually be applied at the inner levels - just what is wanted for time efficiency!)

THE/

THE ENTRY SEQUENCE

This can best be expressed as follows:

```
LDRA     P
STAS     ZX+   /   Save the dynamic pointer

<set  P  to the frame pointer>

<save the static pointer>

<set the stack pointer>

<set up any  MHRs  needed>

PJUMP  <start of procedure body>
```

Here, <set P to the frame pointer> is either:

```
LODP     ZX      }   if  b = 0
```

or

```
SETA     Lb      }   if  b > 0  .
LODP     AX      }
```

<save the static pointer> is either:

```
STBS     P 0
```

or

empty if no static pointer is needed.

The X register is set by <set the stack pointer> which is either:

```
PADD = A = <c+w>  }   if c+w ≥ 16 and b ≠ 0
ADRX   A          }
```

or

```
PGET = A = <c+w>  }   if c+w ≥ 16 and b = 0
ADRX   A          }
```

w is the size of the structure space

or

```
ADMX   L(<c+w>)   }   if c+w < 16
```

or

empty if c+w = 0 .

Only/

Only the MHRs actually used need be set up and <set up any MHRs needed> consists of pairs of instructions of the form:

```
SETA  MO  i
STAS  P   i
```

for each i in 1..d such that Mi is used, but if both Mj and M(j+1) are used it is better to generate the sequence:

```
SETE  MO  j
STES  P   j
```

to replace the four instructions which would otherwise be produced. In the case of the program part no code should be generated for this function, as the MHRs of MAIN are preset in GLOBAL.

THE EXIT SEQUENCE

The coding of the exit sequence depends on whether a procedure or a function is being left.

(1) procedures:

```
LDRA  P
SUBA  L(a+b)
LDRX  A          /  Restore the stack pointer
SETE  P-(b+2)
LODP  ZB         /  Restore the frame pointer
ADMA  L1         /  Update the return address
EXRA  S          /  and exit
```

(2) functions:

```
LDRA  P
SUBA  L(a+b)
LDRX  A          /  Restore the stack pointer
{SETB  P-(b+2)
{LDRY  B         /
                 /
SETE  P-2  }  fetch the single or
  or       }  double length result
SETA  P-2  }  to the accumulator
LODP  P-(b+1)I /  Restore the frame pointer
ADMY  L1       /  Update the return address
EXRY  S        /  and exit
```

REVISED LIST OF ADDRESSING MODES

(1) locals and const (value) parameters:

Pp

/ (2)

(2) var parameters:

Pp I

(3) small constants (<256):

Ll

(4) large constants (>255) and globals:

MmD

(5) locals in the first 64 words of
'scalar space 2' of the enclosing scope:

MO D

(6) all other non-locals:

```
LDRY  P                          /Save frame pointer
{LODP  POI}^n                    / n > 0
<access in a  P  mode>
EXRY  P                          / Restore frame pointer
```

where <access in a P mode> involves either Pp or PpI according as (1) or (2) above is applicable to the object in its own scope.

CASE STATEMENTS

Consider the general case statement:

$$\underline{\text{case}}\ \ e\ \ \underline{\text{of}}\ \ i{:}\ S_i\ ;\ \ldots\ ;\ k{:}\ S_k\ \ \underline{\text{end}}$$

and let e be of type T. Define u as the smallest non-negative value in T and v as the largest case label in the statement. The object code takes the following form:

```
            <evaluate  e  in A>
            PJMP = <L1>
   <Li>     Si
            .
            .
            .
   <Lk>     Sk
   <L1>     PSUB = <u>   }   if  u ≠ 0
            <check case index>
SETB  S2                        /Set B = *+2
SETA  AB                        /Jump back to case  e
JUMP  ZA    (<Lu>)              /This is the switch List
              .
              .
              .
            (<Lv>)
```

where <L1>.... <Lv> are suitably-chosen Usercode labels. The sequence <check case index> generates different code according as checking is required or not. If no checking is required, it produces no code. If checking is required, it should generate:

```
LINK  ZX
PGET = B = <v>
JUMP  SOI
(CASERR)
```

where CASERR is a Usercode routine which returns to the calling routine if and only if $0 \leq (A) \leq (B)$.

ARRAYS OF CHAR

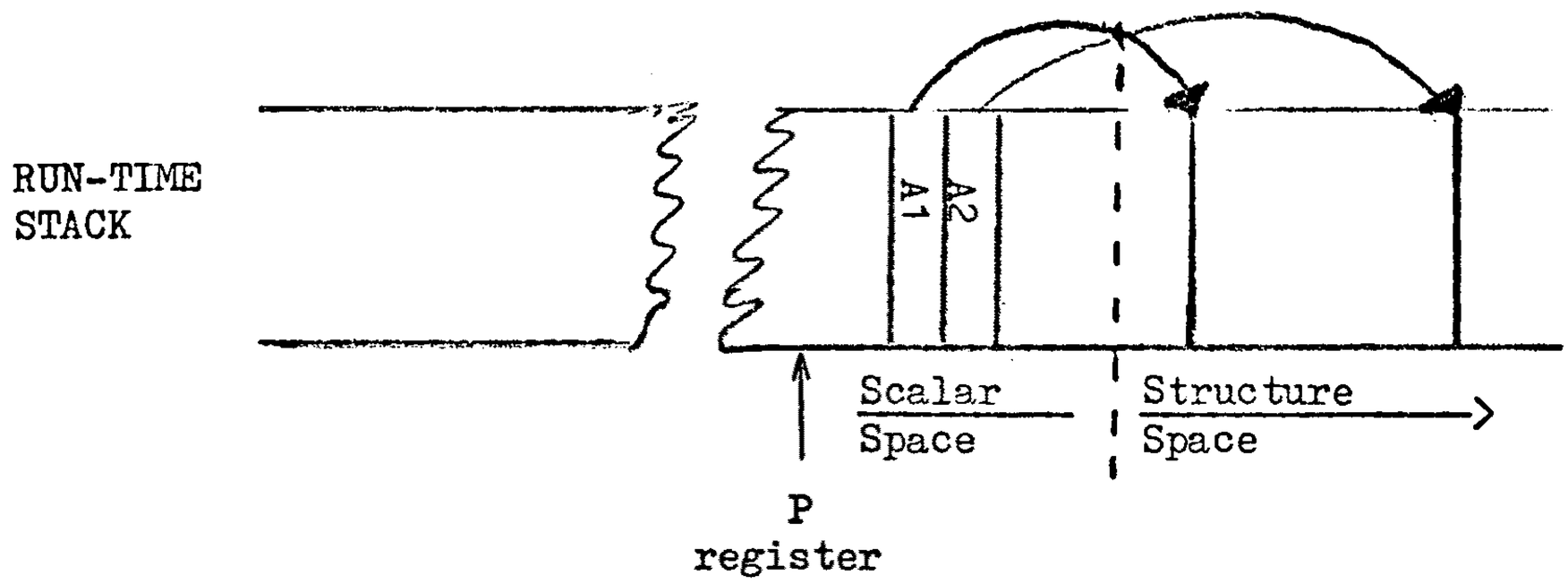
Elements of a char array should occupy only 1 byte (not 1 word). This can be catered for automatically by the addressing modes used to access arrays, if STHS and SETH instructions are used. Simple char variables should occupy the least significant byte of a full word, the top byte being zeroised: thus SETA and STAS instructions must be generated in this case.

Class and Pointer Handling

Introduction

When a procedure (block) containing one or more class declarations is entered, two words in the scalar space (class descriptor) and S words in the structure space of the run-time stack are reserved for each class (where S = maximum number of components of the class * maximum size of the components of the class).

The structure space is allocated to the class components as they are created while the two scalar space words (A1 and A2 say) point to the start of the first (next) free area and the end of the last free area in the structure space for this class:-



Allocation of space to class components

Consider a class C to which pointer PTR is bound. Then, the effect of the procedure call ALLOC (PTR) is to allocate space for a new class component using the address in A1 and to place the address of the space allocated in PTR. The address in A1 is increased by an amount equal to the size of the component.

If there is no space available [contents (A1) + size of component to be allocated > contents (A2)] then a warning message is printed and the value NIL is assigned to PTR.

If a variant component is specified (as a further parameter) in the call of ALLOC, then space just sufficient for the variant is allocated.

Release of class space

a) By the very nature of the run-time stack, class space is automatically released on leaving the procedure in which the class was declared.

b) Some measure of reallocation of class space is allowed by the use of RESET (PTR) where PTR points to some component of the class. Component space allocated beyond the component to which PTR points is reclaimed by copying into A1 the address in PTR.